Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2008, Article ID 594129, 21 pages
doi:10.1155/2008/594129

Research Article

Compilation and Worst-Case Reaction Time Analysis for Multithreaded Esterel Processing

Marian Boldt, Claus Traulsen, and Reinhard von Hanxleden

Department of Computer Science, University of Kiel, 24118 Kiel, Germany

Correspondence should be addressed to Claus Traulsen, ctr@informatik.uni-kiel.de

Received 15 September 2007; Accepted 18 April 2008

Recommended by Michael Mendler

The recently proposed *reactive processing architectures* are characterized by instruction set architectures (ISAs) that directly support reactive control fow including concurrency and preemption. These architectures provide efficient execution platforms for reactive synchronous programs; however, they do require novel compiler technologies, notably with respect to the handling of concurrency. Another key quality of the reactive architectures is that they have very predictable timing properties, which make it feasible to analyze their worst-case reaction time (WCRT). We present an approach to compile programs written in the synchronous language Esterel onto a reactive processing architecture that handles concurrency via priority-based multithreading. Building on this compilation approach, we also present a procedure for statically determining tight, safe upper bounds on the WCRT. Experimental results indicate the practicality of this approach, with WCRT estimates to be accurate within 22% on average.

1. INTRODUCTION

The programming language Esterel [1] has been designed for developing control-dominated reactive software or hardware systems. It belongs to the family of *synchronous languages* [2], which have a formal semantics that abstracts away run-time uncertainties, and allow abstract, well-defined, and executable descriptions of the application at the system level. Hence these languages are particularly suited to the design of safety-critical real-time systems. To express reactive behavior, Esterel offers numerous powerful control flow primitives, in particular concurrency and various preemption operators. Concurrent threads can communicate back and forth instantaneously, with a tight semantics that guarantees deterministic behavior. This is valuable for the designer, but also poses implementation challenges.

Besides being compiled to C and executed as software, or being compiled to VHDL and synthesized to hardware, Esterel can be executed on a *reactive processor* [3]. These processors directly support reactive control flow, such as preemption and concurrency, in their instruction set architecture (ISA). One approach to handle concurrency is multithreading, as implemented in the Kiel Esterel processor (KEP). The KEP uses a priority-based scheduler, which makes threads responsible to manage their own priorities. This scheme allows to keep the scheduler very light-weight. In the KEP, scheduling and context switching do not cost extra instruction cycles, only changing a thread's priority costs an instruction. One challenge for the compiler is to compute these priorities in a way that on the one hand preserves the execution semantics of Esterel and on the other hand does not lead to too many changes of the priorities, since this would decrease the execution speed. We have developed a priority assignment algorithm that makes use of a special concurrent control flow graph and has a complexity that is linear in the size of that graph, which in practice tends to be linear in the size of the program.

Apart from efficiency concerns, which may have been the primary driver towards reactive processing architectures, one of their advantages is their timing predictability. To leverage this, we have augmented our compiler with a timing analysis capability. As we here are investigating the timing behavior for reactive systems, we are specifically concerned with computing the maximal time it takes to compute a single reaction. We refer to this time, which is the time from given input events to generated output events, as *worst-case reaction time* (WCRT). The WCRT determines the maximal rate for the interaction with the environment.

There are two main factors that facilitate the WCRT analysis in the reactive processing context. These are on the one hand the synchronous execution model of Esterel, and on the other hand the direct implementation of this execution model on a reactive processor. Furthermore, these processors are not designed to optimize (average) performance for general purpose computations, and hence do not have a hierarchy of caches, pipelines, branch predictors, and so forth. This leads to a simpler design and execution behavior and further facilitates WCRT analysis. Furthermore, there are reactive processors, such as the KEP, which allow to fix the reaction lengths to a predetermined number of clock cycles, irrespective of the number of instructions required to compute a specific reaction, in order to minimize the jitter.

We here present a WCRT analysis of complete Esterel programs including concurrency and preemption. The analysis computes the WCRT in terms of KEP instruction cycles, which roughly match the number of executed Esterel statements. As part of the WCRT analysis, we also present an approach to calculate potential instantaneous paths, which may be used in compiler analysis and optimizations that go beyond WCRT analysis.

Thus this paper is concerned with both the compilation and the timing analysis of Esterel programs executed on multithreaded reactive processors. Previous reports presented earlier results in both fields [4, 5]. This paper extends and updates these reports, and represents the first comprehensive description of these two closely interrelated areas. Further details can be found in the theses of the first author [6, 7].

In the following section, we consider related work. In Section 3, we will give an introduction into the synchronous model of computation for Esterel and the KEP. We outline the generation of a *concurrent KEP assembler graph* (CKAG), an intermediate graph representation of an Esterel program, which we use for our analysis. Section 4 explains the compilation and Section 5 represents the algorithm for the WCRT analysis. Section 6 presents experimental results that compare the WCRT estimates with values obtained from exhaustive simulation. The paper concludes in Section 7.

2. RELATED WORK

In the past, various techniques have been developed to synthesize Esterel into software; see Potop-Butucaru et al. [8] for an overview. The compiler presented here belongs to the family of simulation-based approaches, which try to emulate the control logic of the original Esterel program directly, and generally achieve compact and yet fairly efficient code. These approaches first translate an Esterel program into some specific graph formalism that represents computations and dependencies, and then generate code that schedules computations accordingly. The EC/Synopsys compiler first constructs a *concurrent control flow graph* (CCFG), which it then sequentializes [9]. Threads are statically interleaved according to signal dependencies, with the potential drawback of superfluous context switches; furthermore, code sections may be duplicated if they are reachable from different control points. The SAXO-RT compiler [10] divides the Esterel program into basic blocks, which schedule each other within the current and subsequent logical tick. An advantage relative to the Synopsis compiler is that the SAXO-RT compiler does not perform unnecessary context switches and largely avoids code duplications; however, the scheduler it employs has an overhead proportional to the total number of basic blocks present in the program. The grc2c compiler [11] is based on the *graph code* (GRC) format, which preserves the state-structure of the given program and uses static analysis techniques to determine redundancies in the activation patterns. A variant of the GRC has also been used in the *Columbia Esterel compiler* (CEC) [12], which again follows SAXO-RT's approach of dividing the Esterel program into atomically executed basic blocks. However, their scheduler does not traverse a score board that keeps track of all basic blocks, but instead uses a compact encoding based on linked lists, which has an overhead proportional to just the number of blocks actually executed.

In summary, there is currently not a single Esterel compiler that produces the best code on all benchmarks, and there is certainly still room for improvements. For example, the simulation-based approaches presented so far restrict themselves to interleaved single-pass thread execution, which in the case of repeated computations ("schizophrenia" [13]) requires code replications.

We differ from these approaches in that we do not want to compile Esterel to C, but instead want to map it to a concurrent reactive processing ISA. Initial reactive ISAs did not consider full concurrency [14, 15] and will not be discussed further here. Since then, two alternatives have been proposed that do include concurrency, namely multiprocessing and multithreading.

The *multiprocessing* approach is represented by the EMPEROR [16], which uses a cyclic executive to implement concurrency, and allows the arbitrary mapping of threads onto processing nodes. This approach has the potential for execution speed-ups relative to single-processor implementations. However, their execution model potentially requires to replicate parts of the control logic at each processor. The EMPEROR Esterel compiler 2 (EEC2) [16] is based on a variant of the GRC, and appears to be competitive even for sequential executions on a traditional processor. However, their synchronization mechanism, which is based on a three-valued signal logic, does not seem able to take compile-time scheduling knowledge into account, and instead repeatedly cycles through all threads until all signal values have been determined.

The *multithreading* approach has been introduced by the Kiel Esterel processor family and has subsequently been adapted by the STARPro architecture [17], a successor of the EMPEROR. The compilation for this type of architecture is a subject of this paper. In some sense, compilation onto KEP assembler is relatively simple, due to the similarities between the Esterel and the KEP assembler. However, we do have to compute priorities for the scheduling mechanism of the KEP, and cannot hard-code the scheduling-mechanism into the generated code directly. Incidentally, it is this dynamic, hardware-supported scheduling that contributes to the efficiency of the reactive processing approach.

It has also been proposed to run Esterel programs on a virtual machine (BAL [18]), which allows a very compact byte code representation. In a way, this execution platform can be considered as an intermediate form between traditional software synthesis and reactive processing; it is a software running on traditional processors, but uses a more abstract instruction set. The proposal by Plummer et al. also uses a multithreaded concurrency model, as in the KEP platform considered here. However, they do not assume the existence of a run-time scheduler, but instead hand control explicitly over between threads. Thus their scheduling problem is related to ours, but does not involve the need to compute priorities as we have to do here. Instead, they have to insert explicit points for context switches. The main difference in both approaches is that the KEP only switches to active threads, while the BAL switches to statically defined control points. One could, however, envision a virtual machine that has an ISA that adopts our multithreading model (a straightforward, albeit inefficient VM would be a KEP simulator), and for which the approach presented here could be applied.

One of the byproducts of our compilation approach is dead code elimination (DCE), see also Section 4.3. Our approach here is rather conservative, considering only static reachability. A more aggressive approach to DCE based on Esterel* (an extension of Esterel with a noninstantaneous jump instruction) has been presented by Tardieu and Edwards [19]. Their approach, as well as other work that performs reachability analysis as part of constructiveness analysis [20], is more involved than our approach in that they perform an (more or less conservative) analysis of the reachable state space.

Regarding timing analysis, there exist numerous approaches to classical worst-case execution time (WCET) analysis. For surveys see, for example, Puschner and Burns [21] or Wilhelm et al. [22]. These approaches usually consider (subsets) of general purpose languages, such as C, and take information on the processor designs and caches into account. It has long been established that to perform an exact WCET analysis with traditional programming languages on traditional processors is difficult, and in general not possible for Turing-complete languages. Therefore WCET analysis typically impose fairly strong restrictions on the analyzed code, such as a-priori known upper bounds on loop iteration counts, and even then control flow analysis is often overly conservative [23, 24]. Furthermore, even for a linear sequence of instructions, typical modern architectures make it difficult to predict how much time exactly the execution of these instructions consumes, due to pipelining, out-of-order execution, argument-dependent execution times (e.g., particularly fast multiply-by-zero), and caching of instructions and/or data [25]. Finally, if external interrupts are possible or if an operating system is used, it becomes even more difficult to predict how long it really takes for an embedded system to react to its environment. Despite the advances already made in the field of WCET analysis, it appears that most practitioners today still resort to extensive testing plus adding a safety margin to validate timing characteristics. To summarize, performing conservative yet tight WCET analysis appears by no means trivial and is still an active research area.

Whether WCRT can be formulated as a classical WCET problem or not depends on the implementation approach. If the implementation is based on sequentialization such that there exist two dedicated points of control at the beginning and the end of each reaction, respectively, then WCRT can be formulated as WCET problem; this is the case, for example, if one "automaton function" is synthesized, which is called during each reaction. If, however, the implementation builds on a concurrent model of execution, where each thread maintains its own state-of-control across reactions, then WCRT requires not only determining the maximal length of predefined instruction sequences, as in WCET, but one also has to analyze the possible control point pairs that delimit these sequences. Thus, WCRT is more elementary than WCET in the sense that it considers single reactions, instead of whole programs, and at the same time WCRT is more general than WCET in that it is not limited to predefined control boundaries.

One step to make the timing analysis of reactive applications more feasible is to choose a programming language that provides direct, predictable support for reactive control flow patterns. We argue that synchronous languages, such as Esterel, are generally very suitable candidates for this, even though there has been little systematic treatment of this aspect of synchronous languages so far. One argument is that synchronous languages naturally provide a timing granularity at the application level, the *logical ticks* that correspond to system reactions, and impose clear restriction onto what programs may do within these ticks. For example, Esterel has the rule that there cannot be *instantaneous loops*: within a loop body, each statically feasible path must contain at least one tick-delimiting instruction, and the compiler must be able to verify this. Another argument is that synchronous languages directly express reactive control flow, including concurrency, thus lowering the need for an operating system with unpredictable timing.

Logothetis et al. [26, 27] have employed model checking to perform a precise WCET analysis for the synchronous language Quartz, which is closely related to Esterel. However, their problem formulation was different from the WCRT analysis problem we are addressing. They were interested in computing the number of ticks required to perform a certain computation, such as a primality test, which we would actually consider to be a transformational system rather than a reactive system [28]. We here instead are interested in how long it may take to compute a single tick, which can be considered an orthogonal issue.

Ringler [29] considers the WCET analysis of C code generated from Esterel. However, his approach is only feasible for the generation of circuit code [13], which scales well for large applications, but tends to be slower than the simulation-based approach.

Li et al. [15] compute a of sequential Esterel programs directly on the source code. However, they did not address concurrency, and their source-level approach could not

consider compiler optimizations. We perform the analysis on an intermediate level after the compilation, as a last step before the generation of assembler code. This also allows a finer analysis and decreases the time needed for the analysis.

One important problem that must be solved when performing WCRT analysis for Esterel is to determine whether a code segment is reachable instantaneously, or delayed, or both. This is related to the well-studied property of *surface* and *depth* of an Esterel program, that is, to determine whether a statement is instantaneously reachable or not, which is also important for schizophrenic Esterel programs [13]. This was addressed in detail by Tardieu and de Simone [30]. They also point out that an exact analysis of instantaneous reachability has NP complexity. We, however, are not only interested whether a statement can be instantaneous, but also whether it can be noninstantaneous.

3. ESTEREL, THE KIEL ESTEREL PROCESSOR AND THE CONCURRENT KEP ASSEMBLER GRAPH

Next we give a short overview of Esterel and the KEP. We also introduce the CKAG, a graph-representation of Esterel, which is used both for the compilation and the WCRT analysis.

3.1. Esterel

The execution of an Esterel program is divided into logical *instants*, or *ticks*, and communication within or across threads occurs via *signals*. At each tick, a signal is either *present* (emitted) or *absent* (not emitted). Esterel statements are either *transient*, in which case they do not consume logical time, or *delayed*, in which case execution is finished for the current tick. Per default statements are transient, and these include for example emit, loop, present, or the preemption operators. Delayed statements include pause, (nonimmediate) await, and every. Esterel's parallel operator, ||, groups statements in concurrently executed threads. The parallel terminates when all its branches have terminated.

Esterel offers two types of preemption constructs. An *abortion* kills its body when an abortion trigger occurs. We distinguish *strong* abortion, which kills its body immediately (at the beginning of a tick), and *weak* abortion, which lets its body receive control for a last time (abortion at the end of the tick). A *suspension* freezes the state of a body in the instant when the trigger event occurs.

Esterel also offers an exception handling mechanism via the trap/exit statements. An exception is *declared* with a trap scope, and is *thrown* (*raised*) with an exit statement. An exit T statement causes control flow to move to the end of the scope of the corresponding trap T declaration. This is similar to a goto statement, however, there are further rules when traps are nested or when the trap scope includes concurrent threads. If one thread raises an exception and the corresponding trap scope includes concurrent threads, then the concurrent threads are weakly aborted; if concurrent threads execute multiple exit instructions in the same tick, the outermost trap takes priority.

3.1.1. Examples

As an example of a simple, nonconcurrent program consider the module ExSeq shown in Figure 1(a). As the sample execution trace illustrates, the module emits signal R in every instant, until it is aborted by the presence of the input signal I. As this is a *weak* abortion, the abortion body gets to execute (emit R) one last time when it is aborted, followed by an emission of S.

The program ExPar shown in Figure 2(a) introduces concurrency: a thread that emits R and then terminates, and a concurrent thread that emits S, pauses for an instant, emits T, and then terminates are executed in an infinite loop. During each loop iteration, the parallel terminates when both threads have terminated, after which the subsequent loop iteration is started instantaneously, that is, within the same tick.

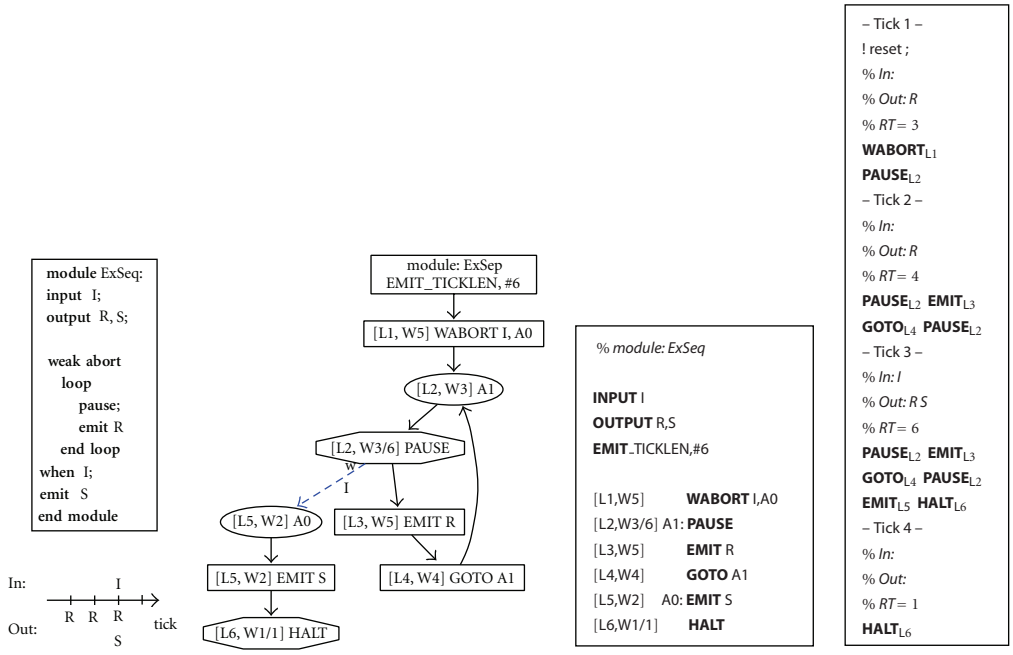
A slightly more involved example is the program Edwards02 [9, 10], shown in Figure 3(a). This program implements the following behavior: whenever the signal S is present, (re-)start two concurrent threads. The first thread first awaits a signal I; it then continuously emits R until A is present, in which case it emits R one last time (weak abortion of the sustain), emits O, and terminates. The second thread tests every other tick for the presence of R, in which case it emits A.

3.1.2. Statement dismantling

At the Esterel level, one distinguishes *kernel statements* and *derived statements*. The derived statements are basically syntactic sugar, built up from the kernel statements. In principle, any set of Esterel statements from which the remaining statements can be constructed can be considered a valid set of kernel statements, and the accepted set of Esterel kernel statements has evolved over time. For example, the halt statement used to be considered a kernel statement, but is now considered to be derived from loop and pause. We here adopt the definition of which statements are kernel statements from the v5 standard [31]. The process of expanding derived statements into equivalent, more primitive statements—which may or may not be kernel statements—is also called *dismantling*. The Esterel program Edwards02-dism (Figure 3(b)) is a dismantled version of Edwards02. It is instructive to compare this program to the original, undismantled version.

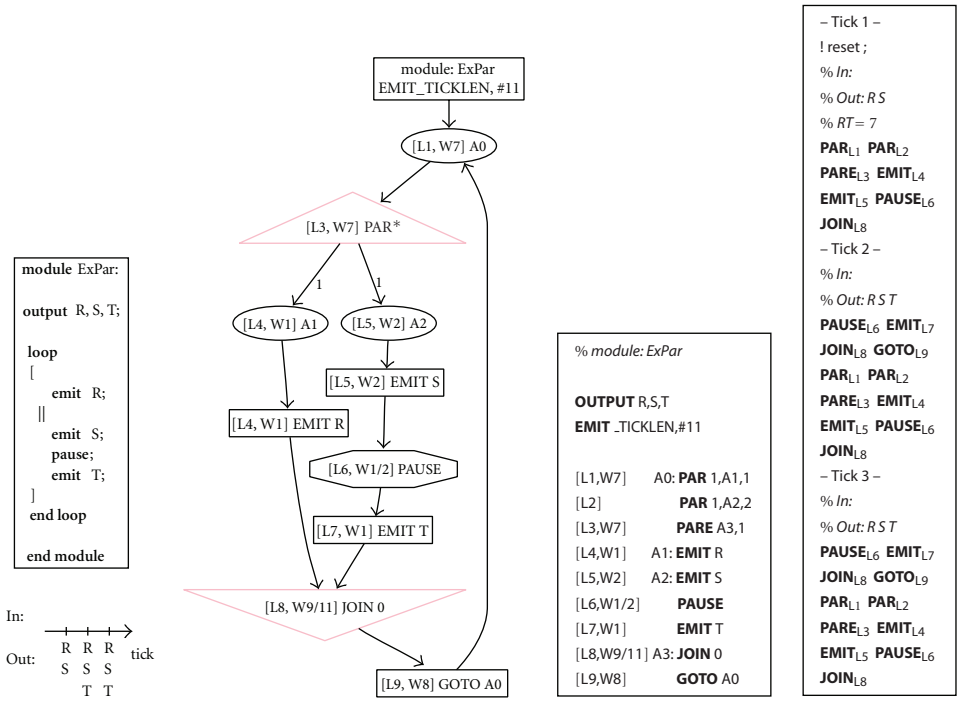
3.2. The Kiel Esterel processor

The instruction set architecture (ISA) of the KEP is very similar to the Esterel language. Part of the KEP instruction set is shown in Table 1; a complete description can be found elsewhere [32]. The KEP instruction set includes all kernel statements (see Section 3.1.2), and in addition some frequently used derived statements. The KEP ISA also includes valued signals, which cannot be reduced to kernel statements. The only parts of Esterel v5 that are not part of the KEP ISA are combined-signal handling and external-task handling, as they both seem to be used only rarely in



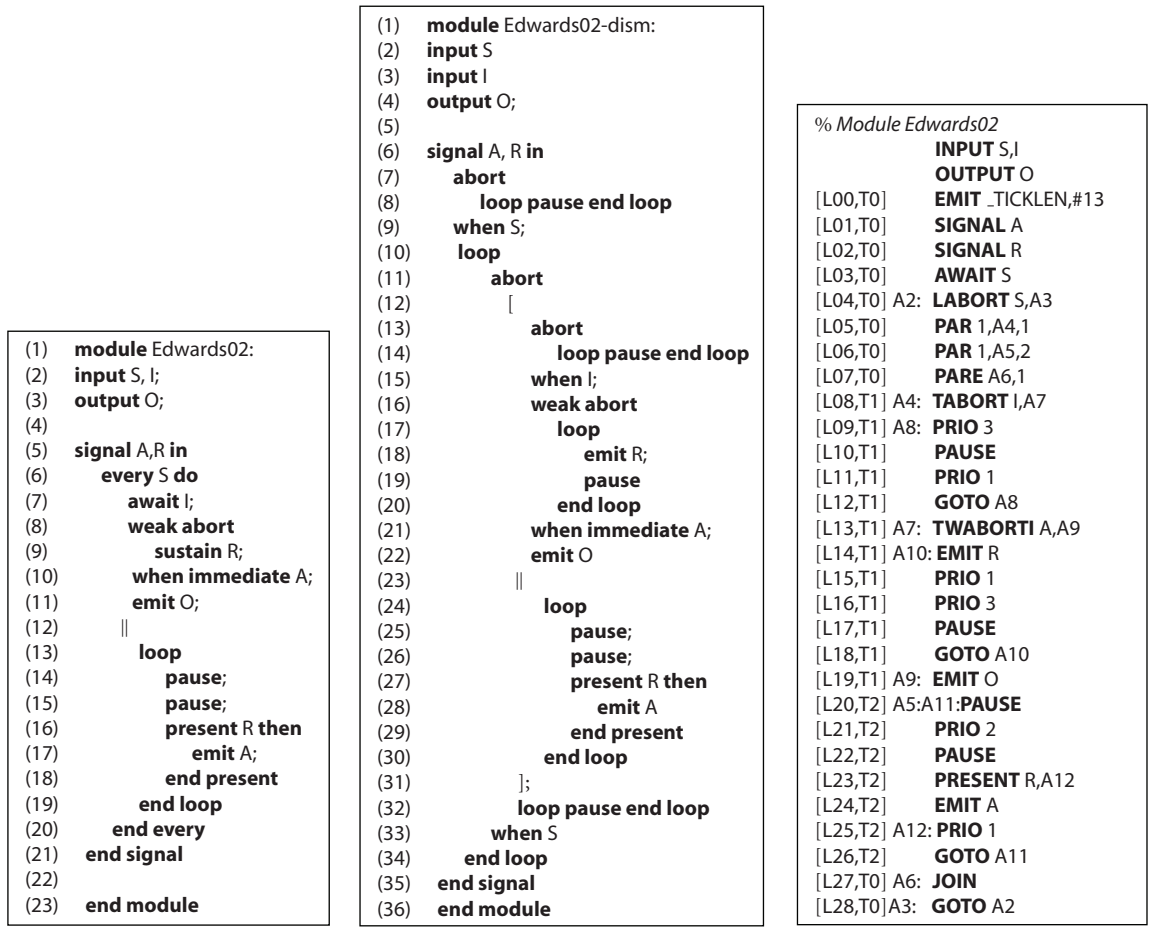
(a) Esterel code and sample trace (b) CKAG (c) KEP assembler (d) Execution trace

Figure 1: A sequential Esterel example. The body of the KEP assembler program (without interface declaration and initialization of the TickManager) is annotated with line numbers L1–L6, which are also used in the CKAG and in the trace to identify instructions. The trace shows for each tick the input and output signals that are present and the reaction time (*RT*), in instruction cycles.



(a) Esterel code and sample trace (b) CKAG (c) KEP assembler (d) Execution trace

Figure 2: A concurrent example program.



```
(1)  module Edwards02:
(2)  input S, I;
(3)  output O;
(4)
(5)  signal A,R in
(6)     every S do
(7)        await I;
(8)        weak abort
(9)           sustain R;
(10)       when immediate A;
(11)       emit O;
(12)    ||
(13)       loop
(14)          pause;
(15)          pause;
(16)          present R then
(17)             emit A;
(18)          end present
(19)       end loop
(20)    end every
(21) end signal
(22)
(23) end module
```

(a) Esterel

```
(1)  module Edwards02-dism:
(2)  input S
(3)  input I
(4)  output O;
(5)
(6)  signal A, R in
(7)     abort
(8)        loop pause end loop
(9)     when S;
(10)    loop
(11)       abort
(12)          [
(13)             abort
(14)                loop pause end loop
(15)             when I;
(16)             weak abort
(17)                loop
(18)                   emit R;
(19)                   pause
(20)                end loop
(21)             when immediate A;
(22)             emit O
(23)          ||
(24)             loop
(25)                pause;
(26)                pause;
(27)                present R then
(28)                   emit A
(29)                end present
(30)             end loop
(31)          ];
(32)          loop pause end loop
(33)       when S
(34)    end loop
(35) end signal
(36) end module
```

(b) Esterel after dismantling

```
% Module Edwards02
                 INPUT S,I
                 OUTPUT O
[L00,T0]         EMIT _TICKLEN,#13
[L01,T0]         SIGNAL A
[L02,T0]         SIGNAL R
[L03,T0]         AWAIT S
[L04,T0] A2:     LABORT S,A3
[L05,T0]         PAR 1,A4,1
[L06,T0]         PAR 1,A5,2
[L07,T0]         PARE A6,1
[L08,T1] A4:     TABORT I,A7
[L09,T1] A8:     PRIO 3
[L10,T1]         PAUSE
[L11,T1]         PRIO 1
[L12,T1]         GOTO A8
[L13,T1] A7:     TWABORTI A,A9
[L14,T1] A10:    EMIT R
[L15,T1]         PRIO 1
[L16,T1]         PRIO 3
[L17,T1]         PAUSE
[L18,T1]         GOTO A10
[L19,T1] A9:     EMIT O
[L20,T2] A5:A11: PAUSE
[L21,T2]         PRIO 2
[L22,T2]         PAUSE
[L23,T2]         PRESENT R,A12
[L24,T2]         EMIT A
[L25,T2] A12:    PRIO 1
[L26,T2]         GOTO A11
[L27,T0] A6:     JOIN
[L28,T0] A3:     GOTO A2
```

(c) KEP assembler

Figure 3: The Edwards02 example [9].

practice. However, adding these capabilities to the KEP ISA seems relatively straightforward.

Due to this direct mapping from Esterel to the KEP ISA, most Esterel statements can be executed in just one instruction cycle. For more complicated statements, well-known translations into kernel statements exist, allowing the KEP to execute arbitrary Esterel programs. The KEP assembler programs corresponding to ExSeq and ExPar and sample traces are shown in Figures 1(c)-1(d) and 2(c)-2(d), respectively, and the KEP assembler program for Edwards02 is shown in Figure 3(c), respectively. Note that PAUSE is executed for at least two consecutive ticks, and consumes an instruction cycle at each tick.

The KEP provides a configurable number of Watcher units, which detect whether a signal triggering a preemption is present and whether the program counter (PC) is in the corresponding preemption body [33]. Therefore, no additional instruction cycles are needed to test for preemption during each tick. Only upon entering a preemption scope two cycles are needed to initialize the Watcher, as for example the WABORT_{L1} instruction in ExSeq (Figure 1(c)) To aid readability, we here use the convention of subscripting KEP instructions with the line number where they occur.

To implement concurrency, the KEP employs a multi-threaded architecture, where each thread has an independent program counter (PC) and threads are scheduled according to their statuses, thread id and dynamically changing priorities: between all active threads, the thread with the highest priority is scheduled. If there is more than one thread with this priority, the highest thread id wins. The scheduler is very light-weight. In the KEP, scheduling and context switching do not cost extra instruction cycles, only changing a thread's priority costs an instruction. The priority-based execution scheme allows on the one hand to enforce an ordering among threads that obeys the constraints given by Esterel's semantics, but on the other hand avoids unnecessary context switches. If a thread lowers its priority during execution but still has the highest priority, it simply keeps executing.

A concurrent Esterel statement with n concurrent threads joined by the ||-operator is translated into KEP assembler as follows. First, threads are *forked* by a series of instructions that consist of n PAR instructions and one PARE instruction. Each PAR instruction creates one thread, by assigning a nonnegative *priority*, a *start address*, and the *thread id*. The end address of the thread is either given implicitly by the start address specified in a subsequent PAR instruction, or, if

Table 1: Overview of the KEP instruction set architecture, and their relation to Esterel and the number of processor cycles for the execution of each instruction.

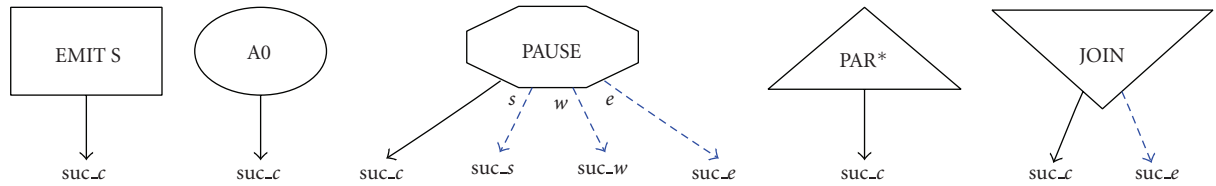
Mnemonic, operands	Esterel syntax	Cycles	Notes
PAR *prio*$_1$, *startAddr*$_1$, *id*$_1$ ... PAR *prio*$_n$, *startAddr*$_n$, *id*$_n$ PARE *endAddr* *startAddr*$_1$: ... *startAddr*$_2$: ... *startAddr*$_n$: ... *endAddr*: JOIN	[p$_1$ ‖ ··· ‖p$_n$]	$n+1$ 1	For each thread, one PAR is needed to define the start address, thread id, and initial priority. The end of a thread is defined by the start address of the next thread, except for the last thread, whose end is dened via PARE. The cycle count of a fork node depends on the count of threads.
PRIO *prio*		1	Set current thread priority to *prio*.
[W]ABORT[I, *n*] *S*, *endAddr* ... *endAddr*:	[**weak**] **abort** ... **when** [**immediate**, *n*] *S*	2	
SUSPEND[I, *n*] *S*, *endAddr* ... *endAddr*:	**suspend** ... **when** [**immediate**, *n*] *S*	2	
startAddr: ... EXIT *exitAddr startAddr* ... *exitAddr*:	**trap** *T* **in** ... **exit** *T* ... **end trap**	1	Exit from a trap, *startAddr/exitAddr* species trap scope. Unlike GOTO, check for concurrent EXITs and terminate enclosing ‖.
PAUSE AWAIT [I, *n*] *S*	**pause** **await** [**immediate**, *n*] *S*	1 1	Wait for a signal. AWAIT TICK is equivalent to PAUSE.
SIGNAL *S*	**signal** *S* **in** ... **end**	1	Initialize a local signal *S*.
EMIT *S* [, {*#data*|*reg*}]	**emit** *S* [(*val*)]	1	Emit (valued) signal *S*.
SUSTAIN *S* [, {*#data*|*reg*}]	**sustain** *S* [(*val*)]	1	Sustain (valued) signal *S*.
PRESENT *S*, *elseAddr*	**present** *S* **then** ... **end**	1	Jump to *elseAddr* if *S* is absent.
HALT	**halt**	1	Halt the program.
addr: ... GOTO *addr*	**loop** ... **end loop**	1	Jump to *addr*.

there is no more thread to be created, it is specified in a PARE instruction. The code block for the last thread is followed by a JOIN instruction, which waits for the termination of all forked threads and concludes the concurrent statement. The example in Figure 2(c) illustrates this: instruction L4 constitutes thread 1, thread 2 spans L5–L8, and the remaining instructions belong to the main thread, which implicitly has id 0.

The priority of a thread is assigned when the thread is created (with the aforementioned PAR instruction), and can be changed subsequently by executing a priority setting instruction (PRIO). A thread keeps its priority across delay instructions; that is, at the start of a tick it resumes execution with the priority it had at the end of the previous tick. This mechanism allows an arbitrary interleaving of thread execution for communicating among threads within the same logical tick. Therefore, a thread may be executed partially, then control may jump to another thread, and later return to the first thread, all within the same tick.

When a concurrent statement terminates, through regular termination of all concurrent threads or via an exception/abort, the priorities associated with the terminated threads also disappear, and the priority of the main thread is restored to the priority upon entering the concurrent statement.

The KEP contains a *TickManager*, which monitors how many instructions are executed in the current logical tick. To minimize jitter, a maximum number of instructions for each logical tick can be specified, via the "special" valued signal _TICKLEN. If the current tick needs less instructions, the start of the next tick is delayed, making the maximum number of instructions the exact number of instructions. If the tick needs more instructions, an error-output is set. Hence a tight, but conservative upper bound of the maximal



(a) transient (b) label (c) delay (d) fork (e) join

Figure 4: Nodes and edges of a concurrent KEP assembler graph.

instructions for one tick, as computed by the WCRT analysis presented in Section 5, is of direct value for the KEP. See Li et al. [15] for details on the TickManager and the relation between the maximum number of instruction per logical tick and the physical timing constraints from the environment perspective.

Note that the KEP compiler per default computes a value for the WCRT and adds a corresponding assembler instruction that specifies a value for _TICKLEN. However, the KEP does not require such a specification of _TICKLEN. If _TICKLEN is left unspecified, the processor "runs freely" and starts the next logical tick as soon as the current tick is finished. This lowers, on average, the reaction time, at the price of a possible jitter.

3.3. The concurrent KEP assembler graph

The CKAG is a directed graph composed of various types of nodes and edges to match KEP program behavior. It is used during compilation from Esterel to KEP assembler, for, for example, priority assigning, dead code elimination, further optimizations, and the WCRT analysis. The CKAG is generated from the Esterel program via a simple structural translation. The only nontrivial aspect is the determination of noninstantaneous paths, which is needed for certain edge types. For convenience, we label nodes with KEP instructions; however, we could alternatively have used Esterel instructions as well.

The CKAG distinguishes the following sets of nodes, see also Figure 4:

L: *label nodes* (ellipses);

T: *transient nodes* (rectangles), which include EMIT, PRESENT, and so forth;

D: *delay nodes* (octagons), which correspond to delayed KEP instructions (PAUSE, AWAIT, HALT, SUSTAIN);

F: *fork nodes* (triangles), corresponding to PAR/PARE;

J: *join nodes* (inverted triangles), corresponding to JOIN;

N: set of all nodes, with $N = T \cup L \cup D \cup F \cup J$.

We also define

A: the *abort nodes*, which denote abortion scopes and correspond to [W]ABORT and SUSPEND; note that $A \subseteq T$.

For each fork node n ($n \in F$), we define

$n.join$: the JOIN statement corresponding to n ($n.join \in J$), and

$n.sub$: the transitive closure of nodes in threads spawned by n.

For abort nodes n ($n \in A$), we define

$n.end$: the end of the abort scope opened by n, and

$n.scope$: the nodes within n's abort scope.

A nontrivial task when defining the CKAG structure is to properly distinguish the different types of possible control flow, in particular with respect to their timing properties (instantaneous or delayed). We define the following types of successors for each n:

$n.suc_c$: the *control successors*. These are the nodes that follow sequentially after n, considering normal control flow without any abortions. For $n \in F$, $n.suc_c$ includes the nodes corresponding to the beginnings of the forked threads.

The successors are statically inserted, based on the syntax of the Esterel program, based on the actual behavior, some of these can be removed. If n is the last node of a concurrent thread, $n.suc_c$ includes the node for the corresponding JOIN—unless n's thread is instantaneous and has a (provably) noninstantaneous sibling thread. Furthermore, the control successors exclude those reached via a preemption ($n.suc_w$, $n.suc_s$)—unless n is an immediate strong abortion node, in which case $n.end \in n.suc_c$.

$n.suc_w$: the *weak abort successors*. If $n \in D$, this is the set of nodes to which control can be transferred immediately, that is when entering n at the end of a tick, via a weak abort; if n exits a trap, then $n.suc_w$ contains the end of the trap scope; otherwise it is $\varnothing$.

If $n \in D$ and $n \in m.scope$ for some abort node m, it is $m.end \in n.suc_w$ in case of a weak immediate abort, or in case of a weak abort if there can (possibly) be a delay between m and n.

$n.suc_s$: the *strong abort successors*. If $n \in D$, these are the nodes to which control can be transferred after a delay, that is when restarting n at the beginning of a tick, via a strong abort; otherwise it is $\varnothing$.

If $n \in D$ and $n \in m.scope$ for some strong abort node m, it is $m.end \in n.suc_s$.

Note that this is not a delayed abort in the sense that an abort signal in one tick triggers the preemption in the next tick. Instead, this means that first a delay has to elapse, and

the abort signal must be present at the next tick (relative to the tick when n is entered) for the preemption to take place.

$n.suc_e$: the *exit successors*. These are the nodes that can be reached by raising an exception.

$n.suc_f$: the *flow successors*. This is the set $n.suc_c \cup n.suc_w \cup n.suc_s$.

For $n \in F$, we also define two kinds of *fork abort successors*. These serve to ensure a correct priority assignment to parent threads in case there is an abort out of a concurrent statement.

$n.suc_{wf}$: the *weak fork abort successors*. This is the union of $m.suc_w \setminus n.sub$ for all $m \in n.sub$ where there exists an instantaneous path from n to m.

$n.suc_{sf}$: the *strong fork abort successors*. This is the set $\cup\{(m.suc_w \cup m.suc_s) \setminus n.sub \mid m \in n.sub\} \setminus n.suc_{wf}$.

In the graphical representation, control successors are shown by solid lines, all other successors by dashed lines, annotated with the kind of successor.

The CKAG is built from Esterel source by traversing recursively over its absract syntax tree (AST) generated by the Colombia Esterel compiler (CEC) [34]. Visiting an Esterel statement results in creating the according CKAG node. A node typically contains exactly one statement, except label nodes containing just address labels and fork nodes containing one PAR statement for each child thread initialization and a PARE statement. When a delay node is created, additional preemption edges are added according to the abortion/exception context.

Note that some of the successor sets defined above cannot be determined precisely by the compiler, but have to be (conservatively) approximated instead. This applies in particular to those successor types that depend on the existence of an instantaneous path. Here it may be the case that for some pair of nodes there does not exist such an instantaneous path, but that the compiler is not able to determine that. In such cases, the compiler conservatively assumes that there may be such an instantaneous path. This is a common limitation of Esterel compilers, and compilers differ in their analysis capabilities here—see also Section 4.1.

4. THE KEP COMPILER

A central problem for compiling Esterel onto the KEP is the need to manage thread priorities during their creation and their further execution. In the KEP setting, this is not merely a question of efficiency or of meeting given deadlines, but a question of correct execution. Specifically, we have to schedule threads in such a fashion that all *signal dependencies* are obeyed. Such dependencies arise whenever a signal is possible emitted and tested in the same tick; we must ensure that all potential emitters for a signal have executed before that signal is tested.

A consequence of Esterel's synchronous model of execution is that there may be *dependency cycles*, which involve concurrent threads communicating back and forth within one tick. Such dependency cycles must be *broken*, for example, by a delay node, because otherwise it would not be possible for the compiler to devise a valid execution schedule that obeys all ordering (causality) constraints. In the Edwards02 example (Figure 3(a)), there is one dependency cycle, from the sustain R_9 instruction in the first parallel thread to the present R_{16} in the second parallel to the emit A_{17} back to the sustain R_9, which is weakly aborted whenever A is present. The dependency cycle is broken in the dismantled version, as there the sustain R has been separated into signal emission (emit R_{18}) and a delay (pause$_{19}$), enclosed in a loop. The broken dependency cycle can also be observed in the CKAG, shown in Figure 5. Referring to nodes by the corresponding line numbers (the "Lxx" part of the node labels) in the KEP assembler code (Figure 3(c)), the cycle is L14 → L23 → L24 → L17 → L18 → L14; it is broken by the delay in L17.

The priority assigned during the creation of a thread and by a particular PRIO instruction is fixed. Due to the nonlinear control flow, it is still possible that a given statement may be executed with varying priorities. In principle, the architecture would therefore allow a fully dynamic scheduling. However, we here assume that the given Esterel program can be executed with a statically determined schedule, which requires the existence of no cyclic signal dependencies. This is a common restriction, imposed for example by the Esterel v7 [35] and the CEC compilers; see also Section 3.3. Note that there are also Esterel programs that are causally correct (*constructive* [1]), yet cannot be executed with a static schedule and hence cannot be directly translated into KEP assembler using the approach presented here. However, these programs can be transformed into equivalent, acyclic Esterel programs [36], which can then be translated into KEP assembler. Hence, the actual run-time schedule of a concurrent program running on KEP is *static* in the sense that if two statements that depend on each other, such as the emission of a certain signal and a test for the presence of that signal, are executed in the same logical tick, they are always executed in the same order relative to each other, and the priority of each statement is known in advance. However, the run-time schedule is *dynamic* in the sense that due to the nonlinear control flow and the independent advancement of each program counter, it in general cannot be determined in advance which code fragments are executed at each tick. This means that the thread interleaving cannot be implemented with simple jump instructions. Instead, a run-time scheduling mechanism is needed that manages the interleaving according to the priority and actual program counter of each active thread.

To obtain a more general understanding of how the priority mechanism influences the order of execution, recall that at the start of each tick, all enabled threads are activated, and are subsequently scheduled according to their priorities. Furthermore, each thread is assigned a priority upon its creation. Once a thread is created, its priority remains the same, unless it changes its own priority with a PRIO instruction, in which case it keeps that new priority until it executes yet another PRIO instruction, and so on. Neither the scheduler nor other threads can change a thread's priority. Note also that a PRIO instruction is considered instantaneous. The only noninstantaneous instructions, which delimit the logical

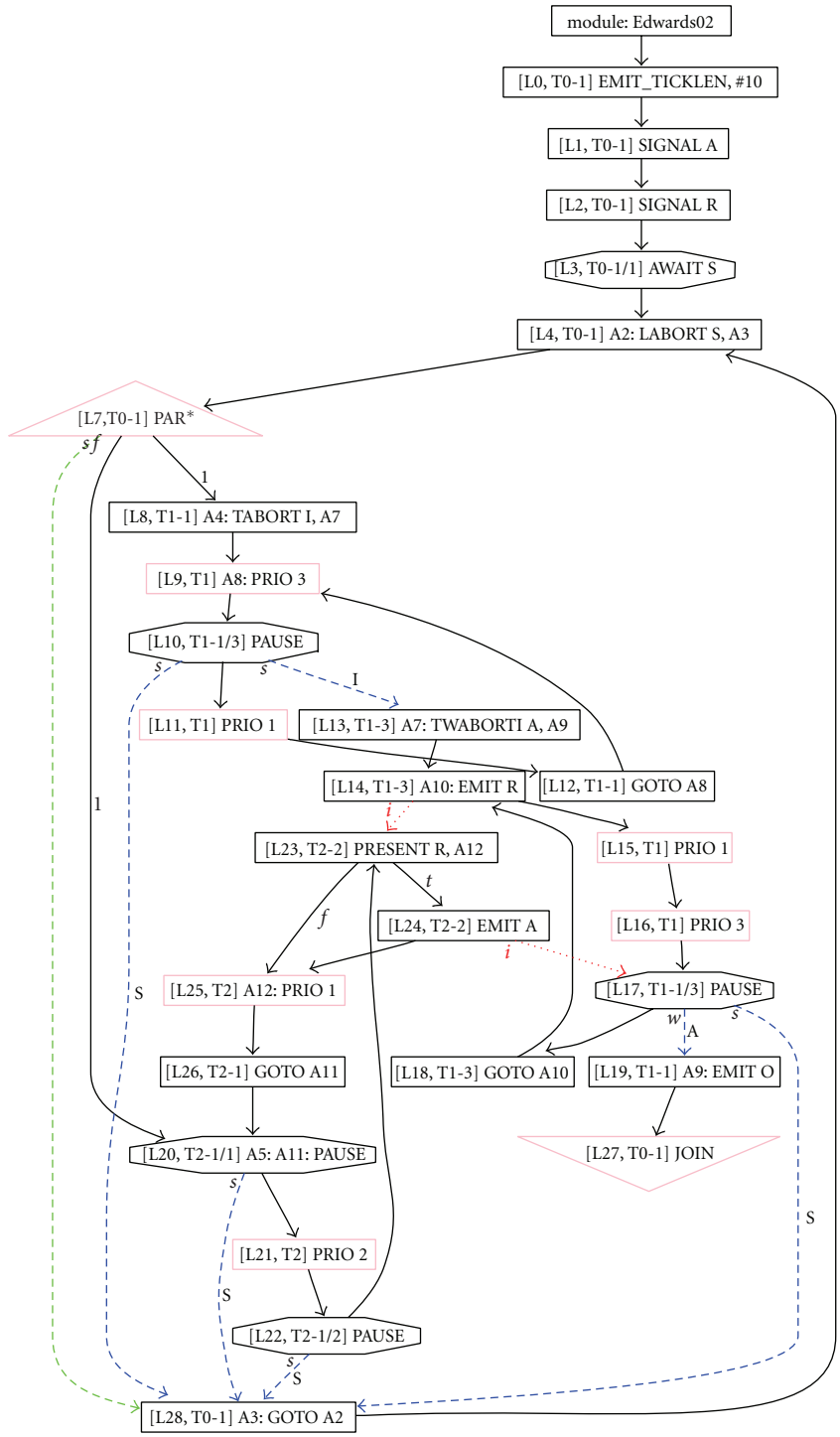


FIGURE 5: The CKAG for the Edwards02 example from Figure 3(a). Dotted lines indicate dependencies (L14 → L23 and L24 → L17), the tail label "i" indicates that these are immediate dependencies (see Section 4.1). For the sake of compactness, label nodes have been incorporated into their (unique) successor nodes.

ticks and are also referred to *delayed* instructions, are the PAUSE instruction and derived instructions, such as AWAIT and SUSTAIN. This mechanism has a couple of implications.

(i) At the start of a tick, a thread is resumed with the priority corresponding to the last PRIO instruction it executed during the preceding ticks, or with the priority it has been created with if it has not executed any PRIO instructions. In particular, if we must set the priority of a thread to ensure that at the beginning of a tick the thread is resumed with a certain priority, it is not sufficient to execute

a PRIO instruction at the beginning of that tick; instead, we must already have executed that PRIO instruction in the preceding tick.

(ii) A thread is executed only if no other active thread has a higher priority. Once a thread is executing, it continues until a delayed statement is reached, or until its priority is lower than that of another active thread or equal to that of another thread with higher id. While a thread is executing, it is not possible for other inactive threads to become active; furthermore, while a thread is executing, it is not possible for other threads to change their priority. Hence, the only way for a thread's priority to become lower than that of other active threads is to execute a PRIO instruction that lowers its priority below that of other active threads.

4.1. Annotating the CKAG with dependencies

In order to compute the thread priorities, we annotate the with additional information about already known priorities and dependencies. For all nodes n, we define

> $n.prio$: the priority that the thread executing n should be running with.

For $n \in D \cup F$, we also define

> $n.prionext$: the priority that the thread executing n should be resumed with in the subsequent tick.

We annotate each node n with the set of nodes that read a signal which is emitted by n. It turns out that analogously to the distinction between *prio* and *prionext*, we must distinguish between dependencies that affect the current tick and the next tick:

> $n.dep_i$: the dependency sinks with respect to n at the current tick (the *immediate dependencies*),
>
> $n.dep_d$: the dependency sinks with respect to n at the next tick (the *delayed dependencies*).

We here assume that the Esterel program given to our compiler has already been established to be causal (constructive), using one of the established constructiveness analysis procedures [20], as for example implemented in the Esterel v5 compiler. We therefore consider only dependencies that cross thread boundaries, as dependencies within a thread do not affect the scheduling. In other words, we assume that intrathread dependencies are already covered by control dependencies; would that not be the case, the program would not be causal, and should be rejected. Should we not want to rely on a separate constructiveness analysis, we would have to consider intrathread dependencies as well.

In general, dependencies are immediate, meaning that they involve statements that are entered at the same tick. An exception are dependencies between emissions of a strong abort trigger signal and corresponding delay nodes within the abort scope, as strong aborts affect control flow at the beginning of a tick and not at the end of a tick. In this case, the trigger signal (say, S) is not tested when the delay node (N) is *entered* as the entering of N marks the end of a tick, and hence control would not even reach N if S was present. However, S is tested when N is *restarted* at beginning of the next tick.

As already mentioned, we assume that the given program does not have cycles. However, what exactly constitutes a cycle in an Esterel program is not obvious, and to our knowledge there is no commonly accepted definition of cyclicity at the language level. The Esterel compilers that require acyclic programs differ in the programs they accept as "acyclic." For example, the CEC accepts some programs that the v5 compiler rejects and vice versa [36], and a full discussion of this issue goes beyond the scope of this paper. Effectively, a program is considered cyclic if it is not (statically) schedulable—and compilers differ in their scheduling abilities. We here consider a program cyclic if the priority assignment algorithm presented in the next section fails. This results in the following definition, based on the CKAG.

Definition 1 (Program cycle). An Esterel program is *cyclic* if the corresponding CKAG contains a path from a node to itself, where for each node n and its successors along that path, n' and n'', the following holds:

$$\begin{aligned} & n \in D \wedge n' \in n.suc_w \\ & \quad \vee\ n \in F \wedge n' \in n.suc_c \cup n.suc_{wf} \\ & \quad \vee\ n \in T \wedge n' \in n.suc_c \cup n.dep_i \\ & \quad \vee\ n \in T \wedge n' \in n.dep_d \wedge n'' \in n'.suc_c \cup n'.suc_s \cup n'.suc_{sf}. \end{aligned} \tag{1}$$

Note that some of the sets that this definition uses are conservatively approximated by the compiler, as already mentioned in Section 3.3. In other words, our compiler may detect spurious cycles and therefore reject a program even if it is causal. As we consider dependencies only if they cross thread boundaries, it appears that we can schedule more programs than other compilers typically can, and we did not encounter a scheduling problem with any of the tested programs. However, a systematic investigation of this issue is still open.

4.2. Computing thread priorities

The task of the priority algorithm is to compute a priority assignment that respects the Esterel semantics as well as the execution model of the KEP. The algorithm computes for each reachable node n in the CKAG the priority $n.prio$ and, for nodes in $D \cup F$, $n.prionext$. According to the Esterel semantics and the observations made in Section 3.3, a correct priority assignment must fulfill the following constraints, where m, n are arbitrary nodes in the CKAG.

Constraint 1 (Dependencies). A thread executing a dependency source node must have a higher priority than the corresponding sink. Hence, for $m \in n.dep_i$, it must be $n.prio > m.prio$, and for $m \in n.dep_d$, it must be $n.prio > m.prionext$.

Constraint 2 (Intratick priority). Within a logical tick, a thread's priority cannot increase. Hence, for $n \in D$ and

```
(1)  procedure main()
(2)    forall n ∈ N do
(3)      n.prio := −1
(4)    V_prio := ∅
(5)    V_prionext := ∅
(6)    N_ToDo := n_root
(7)    while ∃n ∈ N_ToDo \ V_prio do
(8)      getPrio(n)
(9)    forall n ∈ ((D ∪ F) ∩ V_prio) \ V_prionext do
(10)     getPrioNext(n)
(11)  end
```

```
(1)  function getPrioNext(n)
(2)    if n.prionext = −1 then
(3)      if (n ∈ V_prionext) then
(4)        error ("Cycle detected!")
(5)      V_prionext ∪= n
(6)      if n ∈ D then
(7)        n.prionext := prioMax(n.suc_c ∪ n.suc_s)
(8)      elif n ∈ F then
(9)        n.prionext :=
(10)           max(n.join.prio, prioMax(n.suc_sf))
(11)     end
(12)   end
(13)   return n.prionext
(14) end
```

```
(1)  function prio [Next]Max(M)
(2)    p := 0
(3)    forall n ∈ M do
(4)      p := max(p, getPrio[Next](n))
(5)    return p
(6)  end
```

```
(1)  function getPrio(n)
(2)    if n.prio = −1 then
(3)      if (n ∈ V_prio) then
(4)        error ("Cycle detected!")
(5)      V_prio ∪= n
(6)      if n ∈ D then
(7)        n.prio := prioMax(n.suc_w),
(8)        N_ToDo = n.suc_c ∪ n.suc_s
(9)      elif n ∈ F then
(10)       n.prio := prioMax(n.suc_c ∪ n.suc_wf),
(11)       N_ToDo ∪= n.suc_sf ∪ n.join.prio
(12)     elif n ∈ T then
(13)       n.prio := max(prioMax(n.suc_c),
(14)         prioMax(n.dep_i) + 1
(15)         prioNextMax(n.dep_d) + 1)
(16)     end
(17)   end
(18)   return n.prio
(19) end
```

Figure 6: Algorithm to compute priorities.

$m \in n.suc_w$, or $n \in F$ and $m \in n.suc_c \cup n.suc_{wf}$, or $n \in T$ and $m \in n.suc_c$, it must be $n.prio \geq m.prio$.

Constraint 3 (Intertick priority for delay nodes). To ensure that a thread resumes computation from some delay node n with the correct priority, $n.prionext \geq m.prio$ must hold for all $m \in n.suc_c \cup n.suc_s$.

Constraint 4 (Intertick priority for fork nodes). To ensure that a main thread that has executed a fork node n resumes computation—after termination of the forked threads—with the correct priority, $n.prionext \geq n.join.prio$ must hold. Furthermore, $n.prionext \geq m.prio$ must hold for all $m \in n.suc_{sf}$.

One could imagine an iterative approach for priority assignment, where all nodes are initially assigned a low priority and priorities are iteratively increased until all constraints are met. However, this would probably be not very efficient, and it would be difficult to validate its correctness and its termination. As it turns out, there is a better alternative. We can order the computations of all priorities such that when a specific priority value is computed, all the priorities that this value may depend on have already been computed. The algorithm shown in Figure 6 achieves this by performing recursive calls that traverse the CKAG in a specific manner.

The algorithm starts in main(), which, after some initializations, in line 8 calls getPrio() for all nodes that must yet be processed. This set of nodes, given by $N_{\text{ToDo}} \setminus V_{\text{prio}}$ (for "Visited"), initially just contains the root of the CKAG. After *prio* has been computed for all reachable nodes in the CKAG, a forall loop computes *prionext* for reachable delay/fork nodes that have not been computed yet.

getPrio() first checks whether it has already computed $n.prio$. If not, it then checks for a recursive call to itself (lines 3/4, see also Lemma 1). The remainder of getPrio() computes $n.prio$ and, in case of delay and fork nodes, adds nodes to the N_{ToDo} list. Similarly getPrioNext() computes $n.prionext$.

Lemma 1 (Termination). *For a valid, acyclic Esterel program, getPrio() and getPrioNext() terminate. Furthermore, they do not generate a "Cycle detected!" error message.*

Proof (Sketch). getPrio() produces an error (line 4) if it has not computed $n.prio$ yet (checked in line 2) but has already been called (line 3) earlier in the call chain. This means that it has called itself via one of the calls to prioMax() or prioNextMax() (via getPrioNext()). An inspection of the calling pattern yields that an acyclic program in the sense of Definition 1 cannot yield a cycle in the recursive call chain. Since the number of nodes is finite, both algorithms terminate. □

Lemma 2 (Fulfillment of constraints). *For a valid, acyclic Esterel program, the priority assignment algorithm computes an assignment that fulfills Constraints 1–4.*

Proof (Sketch). First observe that—apart from the initialization in main()—each $n.prio$ is assigned only once. Hence, when prioMax() returns the maximum of priorities for a given set of nodes, these priorities do not change any more. Therefore, the fulfillment of Constraint 1 can be

```
(1)  procedure genPrioCode()
(2)    forall n ∈ F do // Step 1
(3)      forall m ∈ n.suc_c do
(4)        annotate corresponding PAR statement with m.prio
(5)
(6)    forall n ∈ N do // Step 2
(7)      // Case p.prio < n.prio impossible !
(8)      P := {p | n ∈ p.suc_f, p.id = n.id}//id is the thread id
(9)      prio := max({p.prio | p ∈ P} ∪ {p.prionext | p ∈ P ∩ D})
(10)     if p. prio > n. prio then
(11)       insert "PRIO␣ n.prio" at n
(12)       // If n ∈ D: insert before n (e.g., PAUSE)
(13)       // If n ∈ T: insert after n (e.g., a label)
(14)
(15)   forall n ∈ D ∪ F do // Step 3
(16)     // Case n. prio > n. prionext is already covered in Step 2
(17)     if n. prio < n. prionext then
(18)       insert "PRIO␣ n.prionext" before n
(19) end
```

Figure 7: Algorithm to annotate code with priority settings according to CKAG node priorities.

deduced directly from getPrio(). Similarly for Constraint 2. Analogously getPrioNext() ensures that Constraints 3 and 4 are met. □

Lemma 3 (Linearity). *For a CKAG with $\mathcal{N}$ nodes and $\mathcal{E}$ edges, the computational complexity of the priority assignment algorithm is $\mathcal{O}(\mathcal{N} + \mathcal{E})$.*

Proof (Sketch). Apart from the initialization phase, which has cost $\mathcal{O}(\mathcal{N})$, the cost of the algorithm is dominated by the recursive calls to getPrio(). The total number of calls is bounded by $\mathcal{E}$. With an amortization argument, where the costs of each call are attributed to the callee, it is easy to see that the overall cost of the calls is $\mathcal{O}(\mathcal{E})$. □

Note also that while the size of the CKAG may be quadratic in the size of the corresponding Esterel program in the worst case, it is in practice (for a bounded abort nesting depth) linear in the size of the program, resulting in an algorithm complexity linear in the program size as well; see also the discussion in Section 6.2.

After priorities have been computed for each reachable node in the CKAG, we must generate code that ensures that each thread is executed with the computed priority. This task is relatively straightforward, Figure 7 shows the algorithm.

Another issue is the computation of thread ids, as these are also considered in scheduling decisions in case there are multiple threads of highest priority. This property is exploited by the scheduling scheme presented here, to avoid needless cycles. The compiler assigns increasing ids to threads during a depth-first traversal of the thread hierarchy; this is required in certain cases to ensure proper termination of concurrent threads [4].

4.3. Optimizations

Prior to running the priority/scheduling algorithm discussed before, the compiler tries to eliminate dependencies as much as possible. It does that using two mechanisms. The first is to try to be clever about the assignment of thread ids, as they are also used for scheduling decisions if there are multiple threads that have the highest priority (see Section 3.2). By considering dependencies between different threads, simple dependencies can be solved without any explicit priority changes. The second mechanism is to determine whether two nodes connected via a dependency are executable within the same instant. This is in general a difficult problem to analyze. We here only consider the special case where two nodes share some (least common) fork node, and one node has only instantaneous paths from that fork node, and the other node only not instantaneous paths. In this case, the dependency can be safely removed.

To preserve the signal-dependencies in the execution, additional priority assignments (PRIO statements) might have to be introduced by the compiler. To assure schedulability, the program is completely dismantled, that is, transformed into kernel statements. In this dismantled graph the priority assignments are inserted. A subsequent "undismantling" step before the computation of the WCRT detects specific patterns in the CKAG and collapses them to more complex instructions, such as AWAIT or SUSTAIN, which are also part of the KEP instruction set.

The KEP compiler performs a statement dismantling (see Section 3.1.2) as a preprocessing step. This facilitates code selection and also helps to eliminate spurious dependency cycles, and to hence increase the set of schedulable (accepted) programs, as already discussed in Section 4. After assigning priorities, the compiler tries again to "undismantle" compound statements whenever this is possible. This becomes apparent in the Edwards02 example; the AWAIT S_{L3} (Figure 3(c)) is the undismantled equivalent of the lines 7–9 in Edwards02-dism (Figure 3(b)).

The compiler suppresses PRIO statements for the main thread, because the main thread never runs concurrently to other threads. In the example, this avoids a PRIO 1 statement at label A3.

Furthermore, the compiler performs dead code elimination, also using the traversal results of the priority assignment algorithm. In the Edwards02 example, it determines that execution never reaches the infinite loop in line 32 of Edwards02-dism, because the second parallel thread never terminates normally, and therefore does not generate code for it.

However, there is still the potential for further optimizations, in particular regarding the priority assignment. In the Edwards02 program, one could for example hoist the PRIO 2_{21} out of the enclosing loop, and avoid this PRIO statement altogether by just starting thread T2 with priority 2 and never changing it again. Even more effective would be to start T3 with priority 3, which would allow to undismantle L08–L12 into a single AWAIT.

5. WORST-CASE REACTION TIME ANALYSIS

Given a KEP program, we define its WCRT as the maximum number of KEP cycles executable in one instant. Thus WCRT analysis requires finding the longest instantaneous

```
(1)  int getWcrtSeq(g)        // Compute WCRT for sequential CKAG g
(2)    forall n ∈ N do n.inst := n.next :=⊥ end
(3)    getInstSeq(g.root)
(4)    forall d ∈ D do getNextSeq(d) end
(5)    return max ({g.root.inst} ⋃{d.next : d ∈ D})
(6)  end
```

```
(1)  int getInstSeq(n)        // Compute statements instantaneously reachable from node n
(2)    if n.inst =⊥ then
(3)      if n ∈ T ∪ L then
(4)        n.inst := max {getInstSeq(c) : c ∈ n.suc_c} + cycles(n.stmt)
(5)      elif n ∈ D then
(6)        n.inst := max {getInstSeq(c) : c ∈ n.suc_w ∪ n.suc_e} + cycles(n.stmt)
(7)      fi
(8)    fi
(9)    return n.inst
(10) end
```

```
(1)  int getNextSeq(d)                    // Compute statements instantaneously reachable
(2)    if d.inst =⊥ then                  // from delay node d at tick start
(3)      d.next := max {getInstSeq(c) : c ∈ d.suc_c ∪ d.suc_s} + cycles(d.stmt)
(4)    fi
(5)    return d.next
(6)  end
```

Figure 8: WCRT algorithm, restricted to sequential programs. The nodes of a CKAG g are given by $N = T \cup L \cup D \cup F \cup J$ (see Section 3.3), $g.root$ indicates the first KEP statement. cycles($stmt$) returns the number of instruction cycles to execute $stmt$, see third column in Table 1.

path in the CKAG, where the length metric is the number of required KEP instruction cycles. We abstract from signal relationships and might therefore consider unfeasible executions. Therefore the computed WCRT can be pessimistic. We first present, in Section 5.1, a restricted form of the WCRT algorithm that does not handle concurrency yet. The general algorithm requires an analysis of instant reachability between fork and join nodes, which is discussed in Section 5.2, followed by the presentation of the general WCRT algorithm in Section 5.3.

5.1. Sequential WCRT algorithm

First we present a WCRT analysis of sequential CKAGs (no fork and join nodes). Consider again the ExSeq example in Figure 1(a).

The longest possible execution occurs when the signal I becomes present, as is the case in Tick 3 of the example trace shown in Figure 1(d). Since the abortion triggered by I is weak, the abort body is still executed in this instant, which takes four instructions: PAUSE_{L2}, EMIT_{L3}, the GOTO_{L4}, and PAUSE_{L2} again. Then it is detected that the body has finished its execution for this instant, the abortion takes place, and EMIT_{L5} and HALT_{L6} are executed. Hence the longest possible path takes six instruction cycles.

The sequential WCRT is computed via a depth-first search (DFS) traversal of the CKAG, see the algorithm in Figure 8. For each node n a value $n.inst$ is computed, which gives the WCRT from this node on in the same instant when execution reaches the node. For a transient node, the WCRT is simply the maximum over all children plus its own execution time.

For noninstantaneous delay nodes, we distinguish two cases within a tick: control can *reach* a delay node d, meaning that the thread executing d has already executed some other instructions in that tick, or control can *start* in d, meaning that d must have been reached in some preceding tick. In the first case, the WCRT from d on within an instant is expressed by the $d.inst$ variable already introduced. For the second case, an additional value $d.next$ stores the WCRT from d on within an instant; "next" here expresses that in the traversal done to analyze the overall WCRT, the $d.next$ value should not be included in the current tick, but in a next tick. Having these two values ensures that the algorithm terminates in the case of noninstantaneous loops: to compute $d.next$ we might need the value $d.inst$.

For a *delay node*, we also have to take abortions into account. The handlers (i.e., their continuations—typically the end of an associated abort/trap scope) of weak abortions and exceptions are instantaneously reachable, so their WCRTs are added to the $d.inst$ value. In contrast, the handlers of strong abortions cannot be executed in the same instant the delay node is reached, because according to the Esterel semantics an abortion body is not executed at all when the abortion takes place. On the KEP, when a strong abort takes place, the delay nodes where the control of the (still active) threads in the abortion body resides are executed once, and then control moves to the abortion handler. In other words, control cannot move from a delay node d to a (strong) abortion handler when control reaches d, but only when it starts in d. Therefore, the WCRT of the handler of a strong abortion is added to $d.next$, and not to $d.inst$.

We do not need to take a weak abortion into account for $d.next$, because it cannot contribute to a longest path.

An abortion in an instant when a delay node is reached will always lead to a higher WCRT than an execution in a subsequent instant where a thread starts executing in the delay node.

The resulting WCRT for the whole program is computed as the maximum over all WCRTs of nodes where the execution may start. These are the start node and all delay nodes. To take into account that execution might start simultaneously in different concurrent threads, we also have to consider the *next* value of join nodes.

Consider again the example ExSeq in Figure 1. Each node n in the CKAG g is annotated with a label "W⟨*n.inst*⟩" or, for a delay node, a label "W⟨*n.inst*⟩/⟨*n.next*⟩." In the following, we will refer to specific CKAG nodes with their corresponding KEP assembler line numbers L⟨n⟩. It is $g.root$ = L1. The sequential WCRT computation starts initializing the *inst* and *next* values of all nodes to $\perp$ (line 2 in getWcrtSeq, Figure 8). Then getInstSeq(L1) is called, which computes L1.*inst* := max {getInstSeq(L2)} + cycles($\text{WABORT}_{\text{L1}}$). The call to getInstSeq(L2) computes and returns L2.*inst* := cycles(PAUSE_{L2}) + cycles(EMIT_{L5}) + cycles(HALT_{L6}) = 3, hence L1.*inst* := 3 + 2 = 5. Next, in line 4 of getWcrtSeq, we call getNextSeq(L2), which computes L2.*next* := getInstSeq(L3) + cycles(PAUSE_{L2}). The call to getInstSeq(L3) computes and returns L3.*inst* := cycles(EMIT_{L3}) + cycles(GOTO_{L4}) + L2.*inst* = 1 + 1 + 3 = 5. Hence L2.*next* := 5 + 1 = 6, which corresponds to the longest path triggered by the presence of signal I, as we have seen earlier. The WCRT analysis therefore inserts an "EMIT_TICKLEN, #6" instruction before the body of the KEP assembler program to initialize the TickManager accordingly, as can be seen in Figure 1(c).

5.2. Instantaneous statement reachability for concurrent Esterel programs

It is important for the WCRT analysis whether a join and its corresponding fork can be executed within the same instant. The algorithm for instantaneous statement reachability computes for a source and a target node whether the target is reachable instantaneously from the source. Source and target have to be in sequence to each other, that is, not concurrent, to get correct results.

In simple cases like EMIT or PAUSE the sequential control flow successor is executed in the same instant respectively next instant, but in general the behavior is more complicated. The parallel, for example, will terminate instantaneously if all subthreads are instantaneous or an EXIT will be reached instantaneously; it is noninstantaneous if at least one subthread is not instantaneous.

The complete algorithm is presented in detail elsewhere [6]. The basic idea is to compute for each node three potential reachability properties: *instantaneous*, *noninstantaneous*, *exit-instantaneous*. Note that a node might be as well (potentially) *instantaneous* as (potentially) *noninstantaneous*, depending on the signal context. Computation begins by setting the *instantaneous* predicate of the source node to *true* and the properties of all other nodes to *false*. When any property is changed, the new value is propagated to its successors. If we have set one of the properties to *true*, we will not set it to *false* again. Hence the algorithm is monotonic and will terminate. Its complexity is determined by the amount of property changes which are bounded to three for all nodes, so the complexity is $O(3*|N|) = O(|N|)$.

The most complicated computation is the property *instantaneous* of a join node, because several attributes have to be fulfilled for it to be *instantaneous*:

(i) For each thread, there has to be a (potentially) instantaneous path to the join node.

(ii) The predecessor of the join node must not be an EXIT, because EXIT nodes are no real control flow predecessors. At the Esterel level, an exception (exit) causes control to jump directly to the corresponding exception handler (at the end of the corresponding trap scope); this jump may also cross thread-bounderies, in which case all threads that contain the jump until the thread that contains the target of the jump and all their sibling threads terminate.

To reflect this at the KEP level, an EXIT instruction does not jump directly to the exception handler, but first executes the JOIN instructions on the way, to give them the opportunity to terminate threads correctly. If a JOIN is executed this way, the statements that are instantaneously reachable from it are not executed, but control instead moves on to the exception handler, or to another intermediate JOIN. To express this, we use the third property besides *instantaneous* and *noninstantaneous*: *exit-instantaneous*.

Roughly speaking, the *instantaneous* property is propagated via for-all quantifier, *noninstantaneous* and *exit-instantaneous* via existence-quantifier.

Most other nodes simply propagate their own properties to their successors. The delay node propagates in addition its *noninstantaneous* predicate to its delayed successors and *exit nodes* propagate *exit-instantaneous* reachability, when they themselves are reachable instantaneously.

5.3. General WCRT algorithm

The general algorithm, which can also handle concurrency, is shown in Figure 9. It emerges from the sequential algorithm that has been described in Section 5.1 by enhancing it with the ability to compute the WCRT of fork and join nodes. Note that the instantaneous of a join node is needed only by a fork node, all other transient nodes and delay nodes do not use this value for their WCRT. The WCRT of the join node has to be accounted for just once in the instantaneous WCRT of its corresponding fork node, which allows the use of a DFS-like algorithm.

The instantaneous WCRT of a fork node is simply the sum of the instantaneously reachable statements of its subthreads, plus the PAR statement for each subthread and the additional PARE statement.

The join nodes, like delay nodes, also have a *next* value. When a fork-join pair (f, j) could be *noninstantaneous*, we have to compute a WCRT $j.next$ for the next instants

```
(1)  int getWcrt(g)        // Compute WCRT for a CKAG g
(2)     forall n ∈ N do n.inst := n.next :=⊥ end
(3)     forall d ∈ D do getNext(d) end
(4)     forall j ∈ J do getNext(j) end // Visit according to hierarchy (inside out)
(5)     return max ({getInst(g.root)} ⋃{n.next : n ∈ D ∪ J})
(6)  end
```

```
(1)  int getInst(n)        // Compute statements instantaneously reachable from node n
(2)     if n.inst :=⊥ then
(3)        if n ∈ T ∪ L then
(4)           t.inst := max {getInst(c) : c ∈ suc_c \ J} + cycles(n.stmt)
(5)        elif n ∈ D then
(6)           n.inst := max {getInst(c) : c ∈ suc_w ∪ suc_e \ J} + cycles(n.stmt)
(7)        elif n ∈ F then
(8)           n.inst := ∑t∈n.suc_c t.inst + cycles(n.par_stmts) + cycles(PARE)
(9)           prop := reachability(n, n.join) // Compute instantaneous reachability of join from fork
(10)          if prop.instantaneous or prop.exit_instantaneous then
(11)             n.inst+ = getInst(n.join)
(12)          elif prop.non_instantaneous then
(13)             n.inst+ = cycles(JOIN) // JOIN is always executed
(14)          fi
(15)       elif n ∈ J then
(16)          n.inst := max {getInst(c) : c ∈ suc_c ∪ suc_e} + cycles(n.stmt)
(17)       fi
(18)    fi
(19)    return n.inst
(20) end
```

```
(1)  int getNext(n)                          // Compute statements instantaneously reachable
(2)     if n.next :=⊥ then                   // from delay node d at tick start
(3)        if n ∈ D then
(4)        n.next := max {getInst(c) : c ∈ suc_c ∪ suc_s \ J ∧ c.id = n.id} + cycles(n.stmt)
(5)        // handle inter thread successors by their according join nodes:
(6)        for m ∈ {c ∈ suc_c ∪ suc_s \ J : c.id ≠ n.id} do
(7)           j := according join node with j.id = m.id
(8)           j.next = max (j.next , getInst(m) + cycles(m.stmt) + cycles(j.stmt))
(9)        end
(10)       elif n ∈ J then
(11)          prop := reachability(n.fork, n) // Compute reachability predicates
(12)          if prop.non_instantaneous then
(13)             n.next := max ((∑t∈n.fork.suc_c max{m.next : t.id = m.id}) + n.inst , n.next)
(14)          fi
(15)       fi
(16)    fi
(17)    return n.next
(18) end
```

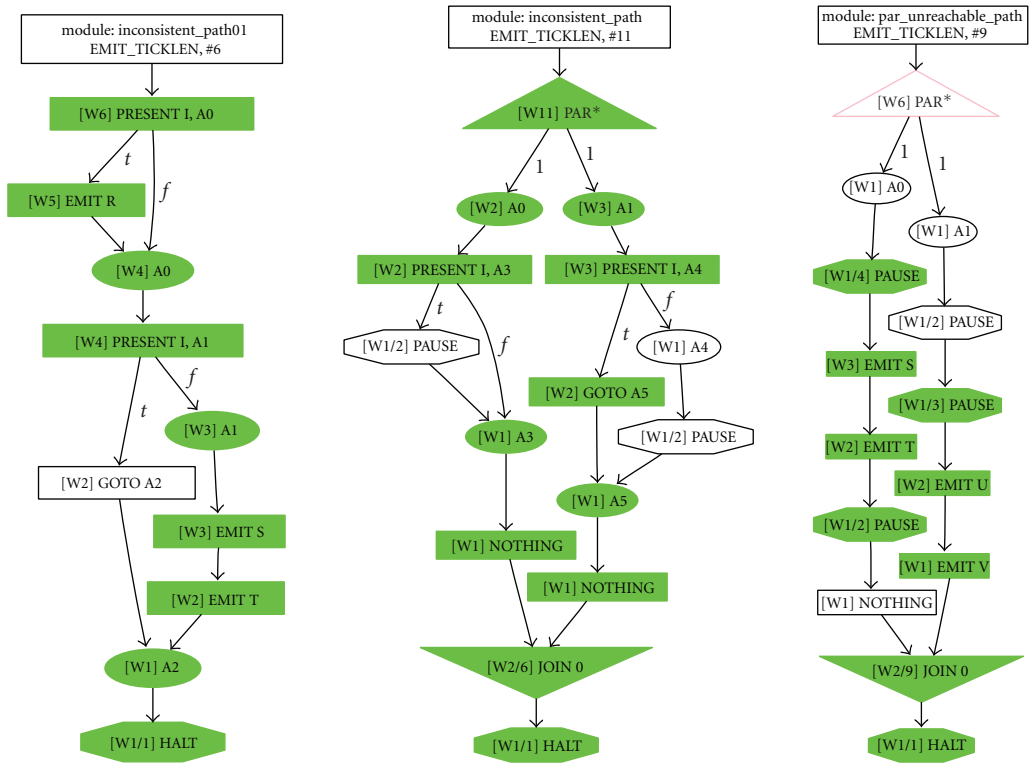
Figure 9: General WCRT algorithm.

analogously to the delay nodes. Its computation requires first the computation of all subthread *next* WCRTs. Note that in case of nested concurrency these *next* values can again result from a join node. But at the innermost level of concurrency the *next* WCRT values all stem from delay nodes, which will be computed before the join *next* values. The delay *next* WCRT values are computed the same way as in the sequential case except that only successors within of the same thread are considered. We call successors of a different thread *interthread-successors* and their WCRT values are handled by the according join node. The join *next* value is the maximum of all interthread-successor WCRT values and the sum of the maximum *next* value for every thread.

If the parallel does not terminate instantaneously, all directly reachable states are reachable in the next instant. Therefore we have to add the execution time for all statements that are instantaneously reachable from the join node.

The whole algorithm computes first the *next* WCRT for all delay and join nodes; it computes recursively all needed *inst* values. Thereafter the instantaneous WCRT for all remaining nodes is computed. The result is simply the maximum over all computed values.

Consider the example in Figure 2(a). First we note that the fork/join pair is always *noninstantaneous*, due to the $\mathsf{PAUSE}_{\mathsf{L6}}$ statement. We compute $\mathsf{L6}.next = \mathsf{cycles}(\mathsf{PAUSE}_{\mathsf{L6}}) + \mathsf{cycles}(\mathsf{EMIT}_{\mathsf{L7}}) = 2$. From the fork node L3, the



(a) Inconsistent sequential path (b) Inconsistent parallel path (c) Unreachable configuration

FIGURE 10: Unreachable path examples.

PAR and PARE statements, the instantaneous parts of both threads and the JOIN are executed, hence L3.*inst* = 2 × cycles(PAR)+cycles(PARE)+cycles(JOIN)+L4.*inst*+L5.*inst* = 2+1+1+1+2 = 7. It turns out that the WCRT of the program is L8.*next* = L6.*next* + L8.*inst* = 2 + 9 = 11. Note that the JOIN statement is executed twice.

A known difficulty when compiling Esterel programs is that due to the nesting of exceptions and concurrency, statements might be executed multiple times in one instant. This problem, also known as *reincarnation*, is handled correctly by our algorithm. Since we compute nested joins from inside to outside, the same statement may effect both the instantaneous and noninstantaneous WCRT, which are added up in the next join. This exactly matches the possible control-flow in case of reincarnation. Even when a statement is executed multiple times in an instant, we compute a correct upper bound for the WCRT.

Regarding the complexity of the algorithm, we observe that for each node its WCRT's *inst* and *next* are computed at most once, and for all fork nodes a fork-join reachability analysis is additionally made, which has itself $O(|N|)$. So we get altogether a complexity of $O(|N|+|D|+|J|)+O(|F|*|N|) = O(2*|N|) + O(|N|^2) = O(|N|^2)$.

5.4. Unreachable paths

Signal informations are not taken into account in the algorithms described above. This can lead to a conservative (too high) WCRT, because the analysis may consider unreachable paths that can never be executed. In Figure 10(a) we see an unreachable path increasing unnecessarily the WCRT because of demanding signal I present and absent instantaneously, which is inconsistent. Nevertheless there is no dead code in the graph, but only two possible paths regarding to path signal predicates.

Figure 10(b) shows an unreachable parallel path that leads to a too high WCRT of the fork node, because the subpaths cannot be executed at the same time. Furthermore, the parallel is declared as possibly instantaneous, even though it is not. Therefore, all statements which are instantaneously reachable from the join node are also added.

Another unreachable parallel path is shown in Figure 10(c). This path is unreachable not because of signal informations but because of instantaneous behavior: the maximal paths of the two threads are never executed in the same instant. In other words, the system is never in a configuration (collection of states) such that both code segments become activated together. Instead of taking for each thread the maximum next WCRT and summing up, it would be more exact to sum up over all threads next WCRT's executable instantaneously and then taking the maximum of these sums. Therefore we would have to enhance the reachability algorithm of the ability to determine how many ticks later a statement could be executed behind another. However, in this case the possible tick counts can become arbitrarily high for each node, so we would get a higher

Table 2: Experimental results for the compiler and priority assignment. For each benchmark it lists the lines of code (LoC) for the source code, the lines of generated KEP assembler, the number of dependencies, the maximal nesting depth of abort scopes, the maximal degree of concurrency, the number of generated PRIO statements, the maximum priority of any thread, and the times for computing the priorities and for the overall compilation.

Esterel		KEP						t_{assign}	t_{comp}
Module name	LoC	Lines	Dependencies	Depth	Max.Conc	#PRIO	Max.Prio	[ms]	[ms]
abcd	152	167	36	2	4	30	3	2.7	14.9
abcdef	232	251	90	2	6	48	3	4.2	63.8
eight_buttons	332	335	168	2	8	66	3	5.9	72.3
channel_protocol	57	61	8	3	4	10	2	0.8	5.3
reactor_control	24	32	5	2	3	0	—	0.4	3.9
runner	26	38	2	2	2	0	—	0.4	4.4
ww_button	94	134	6	3	4	6	2	1.6	10.0
tcint	410	472	65	5	17	45	3	17.3	112.2

complexity and a termination problem. Our analysis is conservative in simply assuming that all concurrent paths may occur in the same instant, and that all can be executed in the same instant as the join.

6. EXPERIMENTAL RESULTS

To evaluate the compilation and WCRT analysis approach presented here, we have implemented a compiler for the KEP based on the CEC infrastructure [34]. We will discuss in turn our validation approach and the quantitative results for the compiler, specifically the priority assignment scheme, and for the WCRT estimation.

6.1. Validation

To validate the correctness of the compilation scheme, as well as of the KEP itself, we have collected a fairly substantial validation suite, currently containing some 500 Esterel programs. These include all benchmarks made available to us, such as the *Estbench* [37], and other programs written to test specific situations and corner cases. An automated regression procedure compiles each program into KEP assembler, downloads it into the KEP, provides an input trace for the program, and records the output at each step. This output is compared to the results obtained from running the same program on a work station, using Esterel Studio.

For each program, any differences in the output traces between the KEP results and the workstation/Esterel Studio results are recorded. Furthermore, the average-case reaction time (ACRT) and WCRT for each program are measured. For these measurements, the KEP is operating in "freely running" mode, that is, _TICKLEN is left unspecified (see Section 3.2); the default would be to set _TICKLEN according to the (conservatively) estimated WCRT,in which case the measured ACRT and WCRT values would be equal to the estimated WCRT. At this point, the full benchmark suite runs through without any differences in output, and the analyzed WCRT is always safe; that is, not lower than the measured WCRT.

Esterel Studio is also used to generate the input trace, using the "full transition coverage" mode. Note that the traces obtained this way still did not cover all possible paths. However, at this point we consider it very probable that a compilation approach that handles all transition coverage traces correctly would also handle the remaining paths. We also feel that this level of validation probably already exceeds the current state of the practice.

6.2. Compilation and priority assignment

As the emphasis here is more on the compilation approach and less on the underlying execution platform, we here refrain from a comparison of execution times and code sizes on the KEP versus traditional, nonreactive platforms; such a comparison can be found elsewhere [4]. Instead, we are here primarily interested in static code characteristics, and in particular how well the priority assignment algorithm works. Table 2 summarizes the experimental results for a selection of programs taken from the *Estbench*.

We note first that the generated code is very compact, and that the KEP assembler line count is comparable to the Esterel source. This is primarily a reflection on the KEP ISA, which provides instructions that directly implement most of the Esterel statements. Furthermore, the relationship between source code and KEP assembler size (and CKAG size) seems fairly linear. We note that the connection between program size and number of (interthread) dependencies is rather loose. For example, eight_buttons is smaller than tcint, but contains more than twice the number of dependencies. Next, we see that the maximal abort nesting depth tends to be limited, only in one case it exceeded three. The degree of concurrency again varied widely; not too surprisingly, the degree of concurrency also influenced the required number of PRIO statements (which—potentially—induce context switches). However, overall the number of generated PRIO statements seems acceptable compared to overall code size, and there were cases where we did not need PRIO at all, despite having several interthread dependencies. This reflects that the thread id assignment mechanism (see Section 4.3) is already fairly efficient in resolving dependencies. Similarly,

the assigned priorities tended to be low in general, for none of the benchmarks they exceeded three. Finally, the priority assignment algorithm and the overall compilation are quite fast, generally in the millisecond range.

6.3. Accuracy of WCRT analysis

As mentioned before, the WCRT analysis is implemented in the KEP compiler, and is used to automatically insert a correct EMIT_TICKLEN instruction at the beginning of the program, such that the reaction time is constant and as short as possible, without ever raising a timing violation by the TickManager. As discussed in Section 6.1, we measured the maximal reaction times and compared it to the computed value. Figure 11 provides a qualitative comparison of estimated and measured WCRT and measured ACRT, more details are given in Table 3. We have never underestimated the WCRT, and our results are on average 22% too high, which we consider fairly tight compared to other reported WCET results [22]. For each program, the lines of code, the computed WCRT and the measured WCRT with the resulting difference are given. We also give the average WCRT analysis time on a standard PC (AMD Athlon XP, 2.2 GHz, 512 KB Cache, 1 GB Main Memory); as the table indicates, the analysis takes only a couple of milliseconds.

The table also compares the ACRT with the WCRT. The ACRT is on average about two thirds of the WCRT, which is relatively high compared to traditional architectures. In other words, the worst case on the KEP is not much worse than the average case, and padding the tick length according to the WCRT does not waste too much resources. On the same token, designing for worst-case performance, as typically must be done for hard real-time systems, does not cause too much overhead compared to the typical average-case performance design. Finally, the table also lists the number of scenarios generated by Esterel-studio and accumulated logical tick count for the test traces.

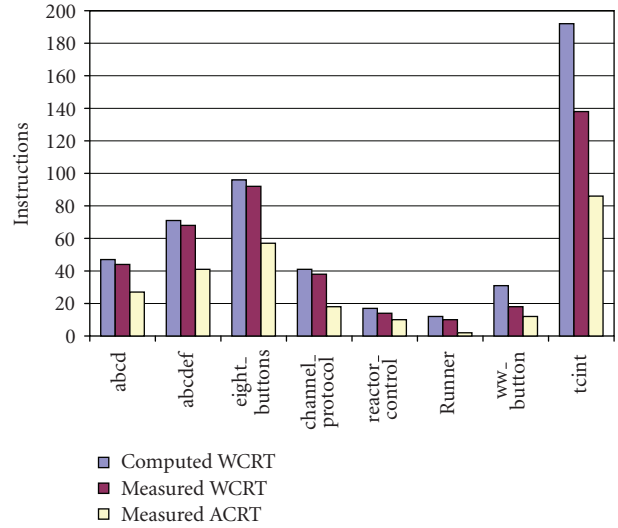


Figure 11: Estimated and measured worst- and average-case reaction times.

7. CONCLUSIONS AND FURTHER WORK

We have presented a compiler for the KEP, and its integrated WCRT analysis. Since the KEP ISA is very similar to Esterel, the compilation of most constructs is straightforward. But the computation of priorities for concurrent threads is not trivial. The thread scheduling problem is related to the problem of generating statically scheduled code for sequential processors, for which Edwards has shown that finding efficient schedules is NP hard [9]. We encounter the same complexity, even though our performance metrics for an optimal schedule are a little different. The classical scheduling problem tries to minimize the number of context switches. On the KEP, context switches are free, because no state variables must be stored and resumed. However, to ensure that a program meets its dependency-implied scheduling constraints, threads must manage their priorities accordingly, and it is this priority switching which contributes to code size and costs an extra instruction at run time. Minimizing priority switches is related to classical constraint-based optimization problems as well as to compiler optimization problems such as loop invariant code motion.

We also have presented the WCRT analysis of Esterel programs. The restricted nature of Esterel and its sound mathematical semantics allow formal analysis of Esterel programs and make the computation of a WCRT for Esterel programs achievable. Our analysis is incorporated in the compiler and uses its internal graph representation, the *concurrent KEP assembler graph* (CKAG). In a first step we compute whether concurrent threads terminate instantaneously, thereafter we are able to compute for each statement how many instructions are maximally executable from it in one logical tick. The maximal value over all nodes gives us the WCRT of the program. The analysis considers concurrency and the multiple forms of preemption that Esterel offers. The asymptotic complexity of the WCRT analysis algorithm is quadratic in the size of the program; however, experimental results indicate that the overhead of WCRT analysis as part of compilation is negligible. We have implemented this analysis into the KEP compiler, and use it to automatically compute an initialization value for the KEP's TickManager. This allows to achieve a high- and constant-response frequency to the environment, and can also be used to detect hardware errors by detecting timing overruns.

Our analysis is safe, that is, conservative in that it never underestimates the WCRT, and it does not require any user annotations to the program. In our benchmarks, it overestimates the WCRT on average by about 22%. This is already competitive with the state of the art in general WCET analysis, and we expect this to be acceptable in most cases. However, there is still significant room for improvement. So far, we are not taking any signal status into account, therefore our analysis includes some unreachable paths. Considering all signals would lead to an exponential growth of the complexity, but some local knowledge should be enough to rule out most unreachable paths of this kind. Also a finer

Table 3: Detailed comparison of WCRT/ACRT times. The WC_e and WC_m data denote the estimated and measured WCRT, respectively, measured in instruction cycles. The ratio $\Delta_{e/m} := WC_e/WC_m - 1$ indicates by how much our analysis overestimates the WCRT. AC_m is the measured average case reaction time (ACRT), AC_m/WC_m gives the ratio to the measured WCRT. Test cases and ticks are the number of different scenarios and logical ticks that were executed, respectively.

Esterel		WCRT			t_{an}	ACRT			
Module name	LoC	WC_e	WC_m	$\Delta_{e/m}$	[ms]	AC_m	AC_m/WC_m	Test cases	Ticks
abcd	152	47	44	7%	1.0	27	61%	161	673
abcdef	232	71	68	4%	1.5	41	60%	1457	50938
eight_buttons	57	41	38	8%	0.4	18	47%	114	556
channel_protocol	57	41	38	8%	0.4	18	47%	114	556
reactor_control	24	17	14	21%	0.2	10	71%	6	20
runner	26	12	10	20%	0.3	2	20%	131	2548
ww_button	94	31	18	72%	1.0	12	67%	8	37
tcint	410	192	138	39%	2.8	86	62%	148	1325

grained analysis of which parts of parallel threads can be executed in the same instant could lead to better results. However, it is not obvious how to do this efficiently.

Our analysis is influenced by the KEP in two ways: the exact number of instructions for each statement and the way parallelism is handled. At least for nonparallel programs our approach should be of value for other compilation methods for Esterel as well, for example, simulation-based code generation. A virtual machine with similar support for concurrency could also benefit from our approach. We would also like to generalize our approach to handle different ways to implement concurrency. A WCRT analysis directly on the Esterel level gives information on the longest possible execution path. Together with a known translation to C, this WCRT information could be combined with a traditional WCET analysis, which takes caches and other hardware details into account.

To conclude, while we think that the approaches for compilation and WCRT analysis presented here are another step towards making reactive processing attractive, there are still numerous paths to be investigated here, including the application of these results towards classical software synthesis. A further issue, which we have not investigated here at all, is to formalize the semantics of reactive ISAs. This would help to deepen the understanding of reactive processing platforms, and could open the door towards formal correctness proofs down to the execution platform. As the ISA provided by the KEP allows to execute programs that are not constructive in the classical sense (such as signal emissions *after* the signals are tested), and yet have a well-defined outcome (i.e., are deterministic), we also envision that this could ultimately lead towards new, interesting synchronous models of computation.

REFERENCES

[1] G. Berry, "The foundations of Esterel," in *Proof, Language and Interaction: Essays in Honour of Robin Milner*, G. Plotkin, C. Stirling, and M. Tofte, Eds., MIT Press, Cambridge, Mass, USA, 2000.

[2] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone, "The synchronous languages 12 years later," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 64–83, 2003.

[3] R. von Hanxleden, X. Li, P. Roop, Z. Salcic, and L. H. Yoong, "Reactive processing for reactive systems," *ERCIM News*, no. 66, pp. 28–29, October 2006.

[4] X. Li, M. Boldt, and R. von Hanxleden, "Mapping esterel onto a multi-threaded embedded processor," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '06)*, pp. 303–314, San Jose, Calif, USA, October 2006.

[5] M. Boldt, C. Traulsen, and R. von Hanxleden, "Worst case reaction time analysis of concurrent reactive programs," in *Proceedings of the Workshop on Model-Driven High-level Programming of Embedded Systems (SLA++P07)*, Braga, Portugal, March 2007.

[6] M. Boldt, *Worst-case reaction time analysis for the KEP3*, Study thesis, Department of Computer Science, Christian-Albrechts-Universität zu Kiel, Kiel, Germany, May 2007, http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/mabo-st.pdf.

[7] M. Boldt, *A compiler for the Kiel Esterel Processor*, Diploma thesis, Department of Computer Science, Christian-Albrechts-Universität zu Kiel, Kiel, Germany, December 2007, http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/mabo-dt.pdf.

[8] D. Potop-Butucaru, S. A. Edwards, and G. Berry, *Compiling Esterel*, Springer, New York, NY, USA, 2007.

[9] S. A. Edwards, "An Esterel compiler for large control-dominated systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, no. 2, pp. 169–183, 2002.

[10] E. Closse, M. Poize, J. Pulou, P. Venier, and D. Weil, "SAXO-RT: interpreting Esterel semantic on a sequential execution structure," *Electronic Notes in Theoretical Computer Science*, vol. 65, no. 5, pp. 80–94, 2002.

[11] D. Potop-Butucaru and R. de Simone, "Optimization for faster execution of Esterel programs," in *Formal Methods and Models for System Design: A System Level Perspective*, pp. 285–315, Kluwer Academic Publishers, Norwell, Mass, USA, 2004.

[12] S. A. Edwards and J. Zeng, "Code generation in the Columbia Esterel compiler," *EURASIP Journal on Embedded Systems*, vol. 2007, Article ID 52651, 31 pages, 2007.

[13] G. Berry, "The constructive semantics of pure Esterel," draft book, 1999, ftp://ftp-sop.inria.fr/esterel/pub/papers/constructiveness3.ps.

[14] P. S. Roop, Z. Salcic, and M. W. S. Dayaratne, "Towards direct execution of Esterel programs on reactive processors," in *Proceedings of the 4th ACM International Conference on Embedded Software (EMSOFT '04)*, pp. 240–248, Pisa, Italy, September 2004.

[15] X. Li, J. Lukoschus, M. Boldt, M. Harder, and R. von Hanxleden, "An esterel processor with full preemption support and its worst reaction time analysis," in *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '05)*, pp. 225–236, ACM Press, San Francisco, Calif, USA, September 2005.

[16] L. H. Yoong, P. Roop, Z. Salcic, and F. Gruian, "Compiling Esterel for distributed execution," in *Proceedings of the International Workshop on Synchronous Languages, Applications, and Programming (SLAP '06)*, Vienna, Austria, March 2006.

[17] S. Yuan, S. Andalam, L. H. Yoong, P. S. Roop, and Z. Salcic, "Starpro—a new multithreaded direct execution platform for esterel," in *Proceedings of the Model Driven High-Level Programming of Embedded Systems Workshop (ETAPS '08)*, Budapest, Hungary, April 2008.

[18] B. Plummer, M. Khajanchi, and S. A. Edwards, "An Esterel virtual machine for embedded systems," in *Proceedings of International Workshop on Synchronous Languages, Applications, and Programming (SLAP '06)*, Vienna, Austria, March 2006.

[19] O. Tardieu and S. A. Edwards, "Approximate reachability for dead code elimination in Esterel," in *Proceedings of the 3rd International Symposium on Automated Technology for Verification and Analysis (ATVA '05)*, pp. 323–337, Taipei, Taiwan, October 2005.

[20] T. R. Shiple, G. Berry, and H. Touati, "Constructive analysis of cyclic circuits," in *Proceedings of the International Design and Test Conference (ITDC '96)*, pp. 328–333, Paris, France, March 1996.

[21] P. Puschner and A. Burns, "A review of worst-case execution-time analysis," *Real-Time Systems*, vol. 18, no. 2-3, pp. 115–128, 2000.

[22] R. Wilhelm, J. Engblom, A. Ermedahl, et al., "The determination of worst-case execution times-overview of the methods and survey of tools," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, 2008.

[23] S. Malik, M. Martonosi, and Y.-T. S. Li, "Static timing analysis of embedded software," in *Proceedings of the 34th Annual Conference on Design Automation (DAC '97)*, pp. 147–152, ACM Press, Anaheim, Calif, USA, June 1997.

[24] A. Burns and S. Edgar, "Predicting computation time for advanced processor architectures," in *Proceedings of the 12th Euromicro Conference on Real-Time Systems (Euromicro-RTS '00)*, pp. 89–96, Stockholm, Sweden, June 2000.

[25] C. Berg, J. Engblom, and R. Wilhelm, "Requirements for and design of a processor with predictable timing," in *Perspectives Workshop: Design of Systems with Predictable Behaviour*, L. Thiele and R. Wilhelm, Eds., vol. 03471 of *Dagstuhl Seminar Proceedings*, Internationales Begegnungs-und Forschungszentrum für Informatik, Schloss Dagstuhl, Germany, 2004.

[26] G. Logothetis and K. Schneider, "Exact high level WCET analysis of synchronous programs by symbolic state space exploration," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE '03)*, pp. 196–203, IEEE Computer Society, Munich, Germany, March 2003.

[27] G. Logothetis, K. Schneider, and C. Metzler, "Exact low-level runtime analysis of synchronous programs for formal verification of real-time systems," in *Forum on Design Languages*, Kluwer Academic Publishers, Frankfurt, Germany, 2003.

[28] D. Harel and A. Pnueli, "On the development of reactive systems," in *Logics and Models of Concurrent Systems*, pp. 477–498, Springer, New York, NY, USA, 1985.

[29] T. Ringler, "Static worst-case execution time analysis of synchronous programs," in *Proceedings of the 5th Ada-Europe International Conference on Reliable Software Technologies (Ada-Europe '00)*, pp. 56–68, Potsdam, Germany, June 2000.

[30] O. Tardieu and R. de Simone, "Instantaneous termination in pure Esterel," in *Proceedings of the 10th International Symposium on Static Analysis Symposium (SAC '03)*, p. 1073, San Diego, Calif, USA, June 2003.

[31] G. Berry, "The Esterel v5 Language Primer, Version v5_91," Centre de Mathématiques Appliquées Ecole des Mines and INRIA, 06565 Sophia-Antipolis, 2000, ftp://ftp-sop.inria.fr/esterel/pub/papers/primer.pdf.

[32] X. Li, *The Kiel Esterel processor: a multi-threaded reactive processor*, Ph.D. thesis, Christian-Albrechts-Universität zu Kiel, Faculty of Engineering, Kiel, Germany, July 2007, http://eldiss.uni-kiel.de/macau/receive/dissertation_diss_00002198.

[33] X. Li and R. von Hanxleden, "A concurrent reactive Esterel processor based on multi-threading," in *Proceedings of the 21st ACM Symposium on Applied Computing (SAC '06)*, vol. 1, pp. 912–917, Dijon, France, April 2006.

[34] S. A. Edwards, "CEC: the Columbia Esterel compiler," 2006, http://www1.cs.columbia.edu/~sedwards/cec/.

[35] Esterel Technologies, Company homepage, http://www.esterel-technologies.com/.

[36] J. Lukoschus and R. von Hanxleden, "Removing cycles in Esterel programs," *EURASIP Journal on Embedded Systems*, vol. 2007, Article ID 48979, 23 pages, 2007.

[37] Estbench Esterel Benchmark Suite, 2007, http://www1.cs.columbia.edu/~sedwards/software/estbench-1.0.tar.gz.